1 Computational Complexity and Problems in P vs. NP

All of the algorithms we have considered so far run in time that grows polynomially with the size of the input. Formally, these algorithms are all $O(n^c)$ for some constant $c \geq 0$, where n is the size of the input. From a general point of view, running in polynomial time is a minimal requirement for calling an algorithm "efficient," and the smaller we make c, the more efficient we say the algorithm is.

Not all problems can be solved in polynomial time, however. For instance, some problems require at least exponential time, running in $O(c^n)$ time, for some constant $c > 1$, and some problems require only poly-logarithmic time, running in $O(\log^c n)$ time. An important class of problems are those that are **NP-hard**, which are problems that *we believe cannot be solved* in polynomial time, even though no one can prove a super-polynomial lower bound.

1.1 Decision vs. Search

As we explore these ideas about the *computational complexity* of different classes of problems, we will consider only different types of *decision problems.* These contrast with the *search problems* we have typically considered so far. Crucially, for any particular problem, there is typically both a decision and a search version:

- *Decision problem*: given an input, a question, and a threshold w, we test for the existence of a solution at least as good as w, answering YES if such a solution exists and NO otherwise.
- *Search problem*: given an input and a question, we return the quality w_* of the best solution possible.

Most of the algorithms we have seen so far are search or optimization problems, because the output of the algorithm is the solution, among all possible solutions, that optimizes some criteria, e.g., maximum flow, minimum spanning tree, etc. When these algorithms succeed, they return a *witness* of the existence of a solution. For instance, in the Minimum Spanning Tree problem, we return the actual MST, from which we can calculate quickly the optimal value w_*. There are some exceptions, of course. For instance, using Bellman-Ford to detect negative weight cycles is an example of a decision problem because it answers the yes/no question of whether G contains a negative weight cycle.

Decision and search problems are intimately related:

every search problem can be turned into a decision problem
by adding a threshold on the quality of the solution.

Consider MSTs again. Instead of asking to find a spanning tree with minimum weight w_* among all spanning trees of G (a search problem), we ask whether there exists a spanning tree whose weight is no more than w (a decision problem). If we set $w < w_*$, then the answer must be NO, because no spanning tree can have weight less than the MST. If we set $w = w_*$, then if an MST exists, then the answer is YES. Moreover, if we set $w > w_*$, then the answer is also YES, because the weight of the MST is always less than w. (It's somewhat beside the point that in this case, there are solutions—spanning trees—that are not a minimum spanning tree).

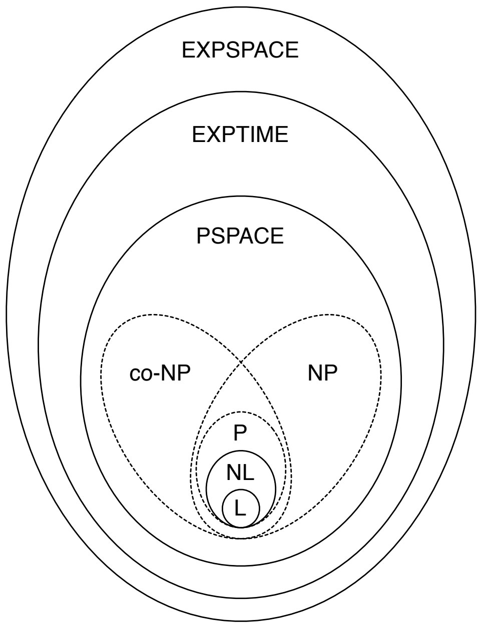
In fact, we can also go the other way:

an algorithm that can solve a particular decision problem
can be used to create a new algorithm that solves the corresponding search problem.

The key here is to run the decision algorithm many times, varying the threshold parameter w in a clever way so that we can find the optimal value w_*. (Do you see how to do this so that it takes no more than $O(\log n)$ calls to the decision algorithm to determine the value $w = w_*$ that solves the search problem?)

1.2 A hierarchy of complexity classes

Because of this relationship, we typically describe computational complexity in terms of decision problems, which we then group into *complexity classes.* A complexity class represents a general statement about the asymptotic minimum time or space required to solve any of its member problems, and we can arrange these classes into a *hierarchy* that formalizes the relative hardness of different problem classes.



Many of the most well-known complexity classes, and their relationship to each other, are shown on the right (dashed lines indicate classes with uncertain relationships).[1] There in the middle is our old familiar friend P, which contains all problems that can be solved in *polynomial time.* Inside P are classes like NL and L, which take strictly less than polynomial time to solve. And containing P are classes that include problems that take strictly more than polynomial time to solve.

[1]There are hundreds of classes known today. The Complexity Zoo website, at the University of Waterloo, is a wonderful resource for understanding the latest results on how they relate to each other. `https://complexityzoo.uwaterloo.ca/`

1.3 Solving vs. verifying (or: can I get a witness?)

In this lecture, we'll focus on the three basic complexity classes:

- **P**: the set of decision problems that can be solved in polynomial time.
- **NP**: the set of decision problems such that if the answer is YES, then there exists a *witness* (succinctly: a proof) of that fact that can be checked in polynomial time.
- **co-NP**: the set of decision problems such that if the answer is NO, then there exists a witness of that fact that can be checked in polynomial time.

Every decision problem that is in P is also in NP, because for every problem in P, we can verify YES answers by simply computing the answer from scratch, in polynomial time.

For instance, sorting a list of numbers is in both P and NP. The decision version of sorting is simple: does there exist a permutation π of the input x such that for all $1 \leq i < n$ the condition $\pi(x_i) \leq \pi(x_{i+1})$ is true? Every sorting algorithm we have seen (QuickSort, MergeSort, HeapSort, InsertionSort, etc.) solves this problem by finding such a permutation π, and each takes polynomial time. Hence, sorting is in P.

Now, suppose that we *guess* a permutation π.[2] With π in hand, it takes $\Theta(n)$ time to verify that $\pi(x)$ is in sorted order. Hence, sorting is in NP. (Sorting is also in co-NP, as is every problem in P. Do you see why?) Notice that for a problem to be in NP, we do not need an algorithm that solves it—we only need to be able to check that an answer handed to us is correct. This is the crucial distinction between P and NP.

The "N" in NP stands *non-deterministic*, which is a mathematical term meaning a theoretical algorithm that non-deterministically[3] arrives at a solution, by making an arbitrary sequence of choices. Hence, the class P represents problems that can be solved in polynomial time, while NP represents problems for which solutions can be verified in polynomial time.

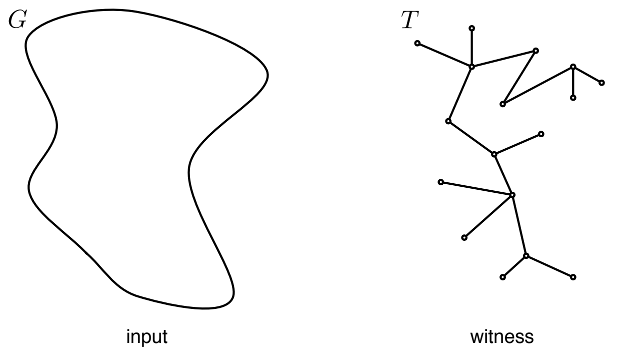
Amazingly, we do not yet know if P and NP are equivalent. Most computer scientists believe that NP is strictly a larger set of problems than P, but we do not have a proof! At a metaphysical level, resolving whether P=NP would reveal whether solving a problem from scratch is fundamentally harder than checking someone else's solution for that problem. At the end of these lecture notes is a table containing many of the problems we've seen in class, but rephrased as decision problems, with a witness.

[2]If you are uncomfortable with the idea of guessing, then imagine someone claims to have developed a new algorithm $\mathcal{A}$ that solves the problem, but they won't reveal the source code. Now, you have black box that, in some way unknown to the outside world, produces π.

[3]Not to be confused with "not deterministic," which refers randomized algorithms.

1.3.1 An example: verifying a witness for MST

To be concrete, let us take the specific problem of finding a weighted spanning tree. Here, the input is an undirected graph $G = (V, E)$ and a weight function on the edges $w : e \to \mathbb{R}$. The decision question is whether there exists a spanning tree T with weight $w(T) \leq k$, where k is a constant. Suppose we (non-deterministically) identify a candidate spanning tree T. How can we verify that $w(T) \leq k$?



Simple. Let T be stored as an adjacency list. For each edge $e \in T$, we add its weight $w(e)$ to an accumulator to compute $w(T) = \sum_{e \in T} w(e)$. By assumption, a call to $w(.)$ takes $O(1)$ time, and we make $|T| = |V| - 1 = n - 1$ calls to compute $w(T)$. Comparing this value with our specified constant k also takes $O(1)$ time. (Finally, we may wish to check that each edge $e \in T$ is also some edge $e \in E$. Even if we are inefficient about how we do this, it takes at worst $|T| \times |E| = O(n^3)$ time. Do you see how we could be more efficient in this step?) Thus, verifying that T is a YES takes time polynomial in the size of the input.

1.4 P vs. NP, and other questions

The crucial difference between P and NP is the difference between solving a problem yourself versus checking that someone else's solution is correct. Perhaps the single most important open question in theoretical computer science, and possibly all of computer science, is whether the classes P and NP are actually different.

An answer to this "P vs. NP" question, i.e., whether P=NP or P≠NP, would have major implications to other branches of science, as well. But, no one knows whether they are the same or different.[4] The Clay Mathematics Institute lists P vs. NP as the first of its Millennium Prize Problems, each of which comes with a \$1,000,000 prize for a correct answer. If you could prove that P=NP and produce an algorithm for solving the hard problems in NP, however, more than half

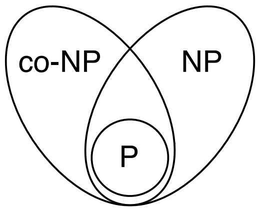
[4]And many have tried. History is littered with false claims one way or another, and new claims are made every year. In fact, it has been proved that deciding whether P=NP is itself a hard question.

of the remaining Millennium Prize Problems would become immediately solvable. Although we do not have a proof of the fact, it is generally believed that P≠NP, but nobody knows how to prove it (any many many have tried).

Intuitively, it certainly *seems* that P and NP are different. Solving a problem from scratch seems and often is difficult. Verifying that someone else's solution is correct seems easier. But the fact that we do not know if P=NP means that we do not, in fact, know whether these two tasks, which seem so different, actually are.

A related, and perhaps more subtle open question is whether NP=co-NP. That is, even if we can quickly verify that YES answers are correct, it does not follow that we can also quickly verify that NO answers are correct. This may seem maddening, but it is an accurate statement: as far as we know, there is no short (polynomial in length) proof or witness that a problem like Traveling Salesman is not satisfiable on a given graph. As a simple analogy, if the YES answer is a needle hiding in a haystack, then I can prove to you that the answer is YES by showing you the needle. But, to prove that the haystack contains NO needles, I have to check all the hay, which could take exponential time.

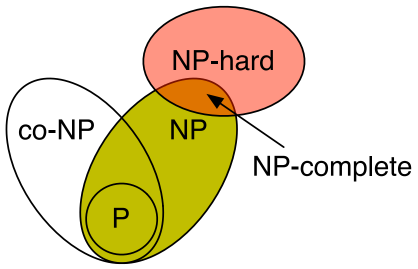
As with P vs. NP, it is generally believed that NP≠co-NP, but, like P vs. NP, no one knows how to prove it. Here is what we think are the relationships between P, NP, and co-NP, although we do not know the proofs:



NP-hard and NP-complete problems. Let us define a problem Q to be **NP-hard** if a polynomial-time algorithm for Q would imply a polynomial-time algorithm for all problems in NP. In this case, a polynomial-time algorithm for Q would imply P=NP. Thus, NP-hard problems are at least as hard as any problem in NP.

A corollary is that the NP-hard set of problems may include some problems that are not, themselves in NP. But, if a problem is NP-hard *and* is a member of NP, then we call it **NP-complete**, which makes it part of the set of hardest problems in NP. That is, a polynomial-time solution to even one NP-complete problem would imply a polynomial-time algorithm to every NP-complete problem. At the time of this writing, thousands of real and interesting problems have been shown to be NP-complete,[5] including several related to the Clay Mathematics Institute's Millennium Prize Problems. Crucially, a polynomial-time solution any one of these problems would yield a polynomial-time solution to all of them, which makes them the most important problems in NP.

Here is another picture of what we think are the relationships between these classes of problems.



How do we know that any problem is in the NP-complete set? Because we have at least one example of a problem that has been proved to be in the NP-complete set:

The Cook-Levin Theorem: BOOLEAN SATISFIABILITY is NP-complete.

To be clear, BOOLEAN SATISFIABILITY is the problem of deciding if a string of boolean variables, chained together with ANDs, ORs and NOTs, has any setting that yields a YES when the operations are carried out.[6] The fact that BOOLEAN SATISFIABILITY (or simply SAT) is NP-complete means that all problems in the NP-complete set can be *reduced*, in polynomial time,[7] to SAT. Thus, if we want to show that some new problem Q is NP-complete, we do not need to prove it directly to be in the set. Instead, we may simply show how to reduce Q to SAT.

All of these statements can be formalized, but we omit the details here.[8]

[5] An excellent, if a little dated, reference for NP-complete problems is Garey and Johnson's excellent book *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman and Company, 1979). Examples of problems you may have heard of that are NP-complete, when they are appropriately generalized, include Tetris (reduction from 3PARTITION), Starcraft (and other RTS, reduction from HAMILTONIAN PATH), Minesweeper (reduction from CIRCUITSAT), and Pac-Man (reduction from HAMILTONIAN CYCLE).

[6] Technically, these are called propositional logic formulas.

[7] Actually, we usually require a logarithmic-space reduction. These are always also polynomial-time reductions, but the reverse is, as far as we know, not true. Nearly all of the standard reductions that we say are polynomial in time are also logarithmic in space, so the difference is usually moot.

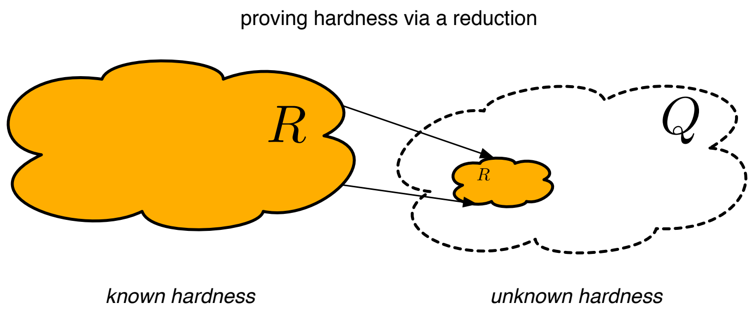
[8] Interested readers should refer to one many excellent texts. Gary and Johnson's book—see footnote 5—is just one. Another is Moore and Mertens, *The Nature of Computation* (Oxford University Press, 2011).

1.5 Reductions

To prove that some problem Q is NP-complete, we use a reduction argument. That is, we demonstrate that another, known hard problem R can be solved *if* we have an algorithm that solves Q. Thus, since we know R is hard, Q must also be hard. Or rather, if Q were easy, then R would also be easy, which contradicts what we know about R.

To prove that a problem Q is NP-complete, show how to reduce a known NP-complete problem to Q.

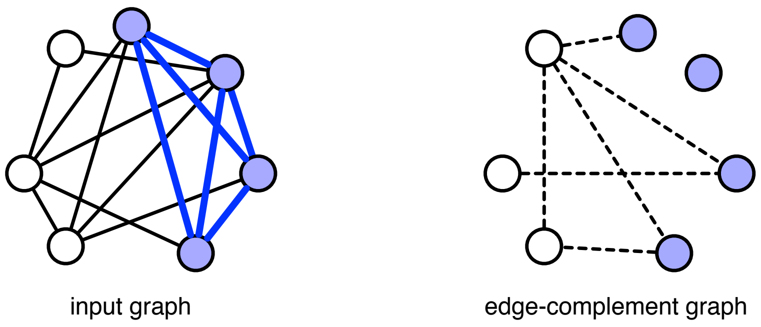
Just as big-O notation is a statement about upper bounds, and just as we are mainly interested in worst-case behavior of our algorithms, the above rule is a statement about how hard some instances are for Q. In every problem, there will be instances that are easy, but these do not determine whether Q is hard *in general.* The hardness of Q is determined by the hardness of its hardest problems. This implies that if we have an algorithm that solves Q, and R is a special case of Q, then our algorithm can solve the hardest cases in R.



That is, a reduction proves that our hard problem R is a special case of our current problem Q, and this proves that Q is at least as hard as R.

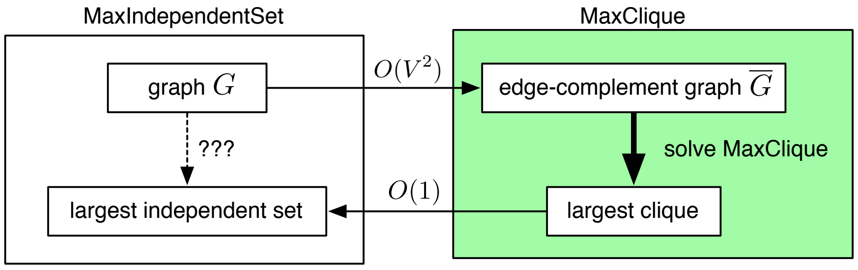
Example 1: Clique **(from** Independent Set**).** Suppose we are given an arbitrary graph $G = (V, E)$. The problem of Max Clique is to compute the number of nodes in the largest clique, which is a subgraph of size $k \leq |V|$ in which every pair of vertices is connected by an edge. That is, find the largest clique in G. The left-hand graph below is a small example G, with the largest clique, of size 4, highlighted; the right-hand figure shows the edge-complement graph $\overline{G}$, with the same vertices highlighted.

To prove that Max Clique is NP-complete, we will reduce it to Max Independent Set, which is known to be NP-complete (by reduction to 3SAT, which is NP-complete by reduction to SAT). Let G be an arbitrary graph. An *independent set* in G is a subset of vertices of G such that for every pair of vertices u, v in the independent set, there is no edge (u, v) in G. The maximum independent



set problem asks to find the largest independent set in G.

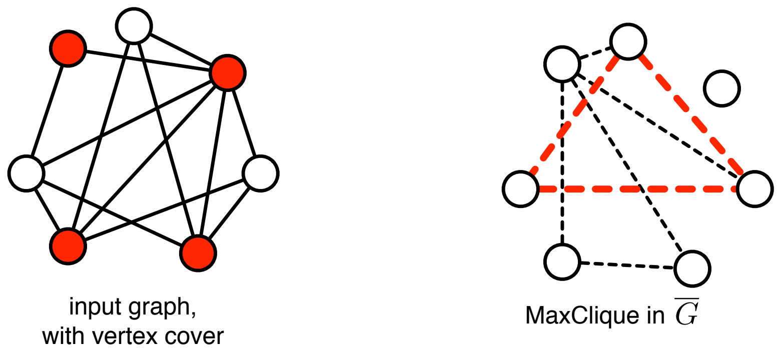
Here is the proof that Max Clique is NP-complete. Let the edge-complement graph $\overline{G} = (V, \overline{E})$ where $\overline{E} = V \times V - E$, i.e., it contains an edge if and only if that edge is not contained in E. Now, a set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$, and we can compute the largest independent set in a graph by computing the largest clique in the edge-complement graph.



Example 2: Vertex Cover **(from** Independent Set**).** Suppose we are again given an undirected graph $G = (V, E)$. The problem of Vertex Cover is to find a set of vertices $C \subseteq V$ such that every edge $e \in E$ is "touched" by one of the vertices in the cover $v \in C$. The Min Vertex Cover problem is to find the smallest vertex cover of G. Here is an example of a vertex cover. To prove that Min Vertex Cover is NP-complete, we make use of the following fact.

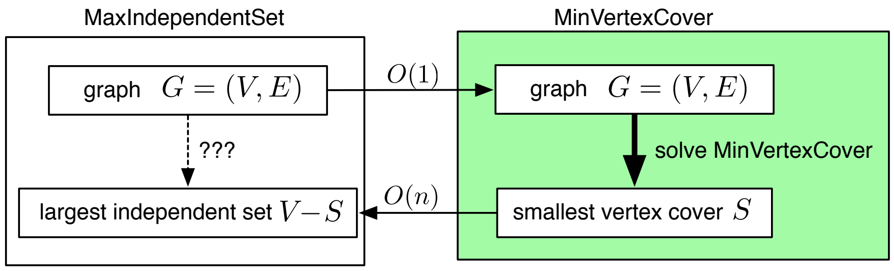
Lemma: If S is an independent set on G, then $V - S$ is a vertex cover.

Proof: Suppose $S \subseteq V$ is an independent set and $(u, v) \in E$. At most one of u, v can be in S, which implies that at least one or the other or both are in $V - S$. Thus, every edge is covered by $V - S$. Similarly, suppose $V - S$ is a vertex cover and $u, v \in S$. Then $(u, v) \notin E$ because otherwise



(u, v) would not be covered by $V - S$, which implies that S is an independent set. □

Thus, to find the largest independent set, we simply need to find the vertices that are not in the smallest vertex cover on the same graph (and vice versa).



Example 3: Traveling Salesman. One of the most famous NP-complete problems is called the TRAVELING SALESMAN PROBLEM, or TSP.[9] In this problem, the input is a collection of n cities and the cost of travel d_{ij} between each pair of cities i, j, which we represent as an $n \times n$ matrix. The output is a *tour* that visits each city exactly once and then returns to the starting city, and we seek the tour with the smallest possible weight.

We can represent the solution to this problem as a permutation $\pi : \{1 \ldots n\} \to \{1 \ldots n\}$ where π_i denotes i successor in the sequence (much like the *pred* function in spanning-tree algorithms). Such a permutation implies a tour if we require that the nth element of π returns us to the 1st

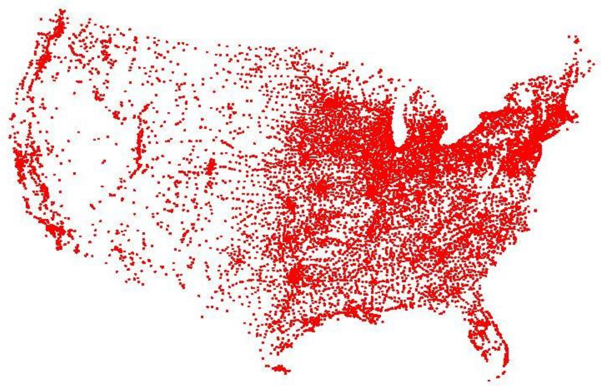
[9]An excellent and comprehensive text on both historical and modern approaches for solving TSP is *The Traveling Salesman Problem* by Applegate, Bixby, Chvátal & Cook (Princeton University Press, 2007). Chapter 1 of this book is available online here: `http://press.princeton.edu/chapters/s8451.pdf`, and has a number of pretty pictures showing TSP solutions for both real and imaginary instances.

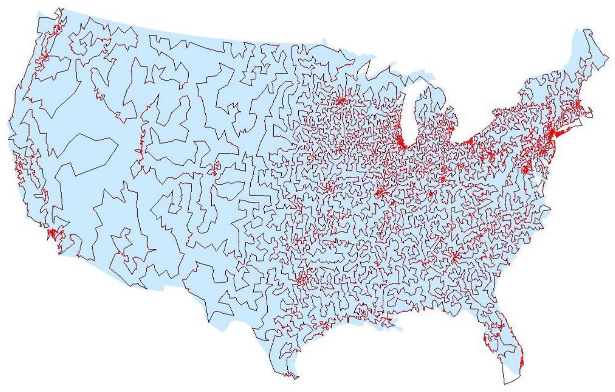
city, so that $1 \to \pi(1) \to \pi(\pi(1)) \to \cdots \to 1$. With this formalization in hand, we can define the optimization and decision versions of TSP:

TSP (optimization)		TSP (decision)	
Input:	an $n \times n$ matrix $d_{ij} \geq 0$	Input:	an $n \times n$ matrix $d_{ij} \geq 0$ and constant k
Output:	a permutation π that minimizes $\text{cost}(\pi) = \sum_{i=1}^{n} d_{i,\pi_i}$	Question:	is there a permutation π with $\text{cost}(\pi) \leq k$?

This formalization makes it immediately clear that there are $n!$ possible tours, and an exhaustive search of them to find the best tour would take exponential time.

A pleasant property of Traveling Salesman problems, particularly those in Euclidean space, is that they make pretty pictures. Here is an example, using the 13509 cities in the contiguous United States that had a population of at least 500 in 1998, showing both the cities on the left and the optimal tour on the right.[10]


13509 cities, USA


13509 city tour

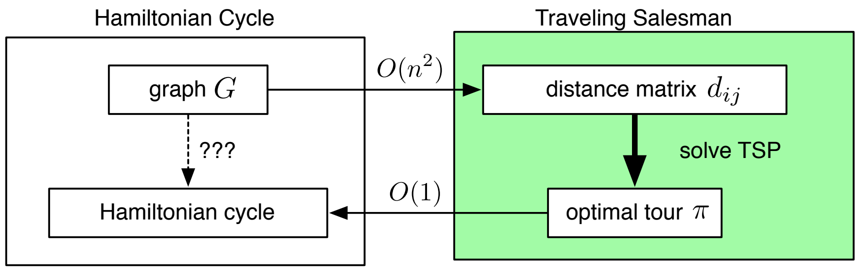
TSP is very similar to another NP-complete problem called Hamiltonian Cycle, in which we are given an unweighted, undirected graph $G = (V, E)$ and we seek a permutation π of the vertices such that each $(i, \pi_i) \in E$. We can prove that TSP is also NP-complete via a reduction from Hamiltonian Cycle by showing how to convert any instance of Hamiltonian Cycle into an instance of TSP.

[10]Images courtesy of William J. Cook at Georgia Tech, `http://www.isye.gatech.edu/~wcook/`, who is one of the authors of the text listed in footnote 9. The solution itself is by those authors. For much more on TSP, see `http://www.math.uwaterloo.ca/tsp/` .

Given G, we define a distance between each pair of vertices as

$$d_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ n & \text{if } (i,j) \notin E \end{cases} .$$

Thus, each pair of vertices that forms an edge in G is connected by a "cheap" edge in our TSP instance, while each pair that is not connected in G is connected by an edge whose cost is greater than the cost of the Hamiltonian cycle (if it exists) in our TSP instance. Under this reduction, which takes $O(n^2)$ time, G has a Hamiltonian cycle if and only if the corresponding TSP instance has a tour of cost n, while if G has no Hamiltonian cycle, $\text{cost}(\pi) \geq 2n - 1$.



2 Approximation algorithms

It may seem hopeless to solve NP-complete problems directly, since we have no polynomial-time algorithms for solving them and the alternative is either exhaustive search or heuristics (which do not provide guarantees, and thus may misbehave badly on some inputs). For optimization problems, there is a third path: *polynomial-time approximation schemes* (PTAS), which provide a guarantee on how far their result is from the optimal solution.

Let A be an algorithm, let x be an instance, let $A(x)$ be the quality of the solution produced by A on x, and let $\text{OPT}(x)$ be the quality of the optimal solution. We say that A is a *ρ-approximation* for a maximization problem if, for all instance x:

$$\frac{A(x)}{\text{OPT}(x)} \geq \rho \ , \tag{1}$$

where $\rho \leq 1$ and we would like ρ to be as large as possible. For a minimization problem, we say

$$\frac{A(x)}{\text{OPT}(x)} \leq \rho \ , \tag{2}$$

where $\rho \geq 1$ and we would like ρ to be as small as possible. It turns out that not all problems are equally hard to approximate, and sometimes we can get ρ to be arbitrarily close to 1, albeit often at the cost of increased running time.

Example 1: Vertex Cover. Recall that in the VERTEX COVER (optimization) problem, we are given a graph $G = (V, E)$ and we see a cover set of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, at least one or both of $u, v \in C$. That is, every edge in G touches a vertex in the cover. Let C_{opt} denote the optimal vertex cover for G.

Here is a very simple 2-approximation algorithm for VERTEX COVER. Let $C = \emptyset$, initially. We then choose an arbitrary edge (u, v) such that $u, v \notin C$, i.e., an edge such that neither of its end points are in the cover. Place both of u, v in C and repeat until there are no such edges. When the algorithm halts, C is a vertex cover. (How long does this algorithm take to run when G is an adjacency list? What about an adjacency matrix?)

How large will C be? First, observe that $|C| = 2k$ if we selected k edges before halting. Since these vertices have no edges in common (do you see why?), and because each edge has at least one endpoint in C_{opt}, we have $|C_{\text{opt}}| \geq k$. Thus, C is at most twice the size of the optimal vertex cover, and

$$\rho = \frac{|C|}{|C_{\text{opt}}|} \leq \frac{2k}{k} = 2 \ . \tag{3}$$

This approximation ratio is tight when G is a complete bipartite graph on $2n$ vertices, where the algorithm selects all $2n$ vertices whereas a cover that consists of one side of the graph contains only n vertices.[11]

Example 2: Metric Traveling Salesman. The general, we do not have algorithms for approximating TSP with any reasonable approximation ratio, and it has been proved that we have no hope of doing so unless P=NP. However, there are variations of TSP, which are still NP-complete, for which good approximation algorithms exist. In particular the METRIC TRAVELING SALESMAN PROBLEM, or ΔTSP, in which we restrict distances obey the triangle inequality

$$d_{ij} \leq d_{ik} + d_{kj} \qquad \text{for all } i, j, k \ , \tag{4}$$

yields constant-factor approximations.[12] Below, we will describe a 2-approximation algorithm for ΔTSP. A 3/2-approximation algorithm, due to Christofides, uses similar ideas.

Claim: the minimum spanning tree T provides a 2-approximation to the optimal ΔTSP tour.

Proof: Let OPT be the cost of the optimal tour on G. We can transform the MST T into a tour by converting each undirected edge in the tree $(u, v) \in T$ into a pair of reciprocated directed edges

[11]A greedy heuristic, in which we repeatedly choose the $v \in V - C$ that would cover the most uncovered edges, actually performs much worse than this approximation algorithm, and has an approximation ratio of $\rho = \Theta(\log n)$. Can you identify a graph G produces this poor performance?

[12]Metric TSP is NP-complete by reduction from Hamiltonian Cycle.

$\{(u,v),(v,u)\}$, and then skipping vertices we have already visited on the tour. There now exists an Eulerian tour of T because every vertex has even degree, and the cost of this tour is an upper bound on OPT:

$$\text{OPT} \leq 2\ell(T) \ , \tag{5}$$

where $\ell(T)$ is the length of the MST T.

Observe also that if we remove a single edge from the optimal tour, we have a spanning tree consisting of a path with $n-1$ edges. This path has length at most OPT, and is at least at long as the MST T, implying the lower bound

$$\ell(T) \leq \text{OPT} \ , \tag{6}$$

where $\ell(T)$ is the length of the Eulerian tour on T. □

Christofides algorithm improves upon this algorithm by recognizing that the only vertices in T that prevent the existence of an Eulerian tour are the odd-degree vertices. Thus, instead of doubling the degree in T of every vertex, it only adds a single edge to each odd-degree vertex. There must be an event number of such odd-degree vertices in T (do you see why?), by creating a perfect matching on these vertices, we can "Eulerify" T.

The cost of our tour, when we skip vertices we have previously visited, is now $\ell \leq \ell(T) + \ell(M)$, where M is the length of the perfect matching on the odd-degree vertices of T. It can be shown that $\ell(M) \leq \text{OPT}/2$ (a fact that depends on the triangle inequality), which implies $\ell \leq \frac{3}{2}\text{OPT}$.

3 On your own

1. Read Chapter 34, NP-Completeness in CLRS
2. Read Chapter 35, Approximation Algorithms CLRS

Problem	**Input**	**Decision**	**Witness**
SORTING	an array of elements A	is there a permutation of A with no more than k out-of-order elements?	a permutation of A
INTERVAL SCHEDULING	a set of requests R and starting/ending functions s and f	is there a compatible set of requests S of size at least k?	a compatible set S
STRING ENCODING	set of input-string symbol frequencies F	is there an encoding C of F such that the encoded string has length $\ell \leq k$?	a codebook C
KNAPSACK	set of items B, capacity W, value function $v : b \to \mathbb{R}$ and weight function $w : b \to \mathbb{R}$	is there a set of items S with weight $w(S) \leq W$ and value $v(S) \geq k$?	a set of items S
STRING ALIGNMENT	strings A and B of lengths n and m	is there an alignment T of A and B with $\text{cost}(T) \leq k$?	an alignment T
SHORT PATHS	directed graph $G = (V, E)$, vertex $s \in V$, and weight function $w : e \to \mathbb{R}$	is there a spanning tree T rooted at s with weight $w(T) \leq k$?	a spanning tree T
WEIGHTED SPANNING TREE	undirected graph $G = (V, E)$ and weight function $w : e \to \mathbb{R}$	is there a spanning tree T with weight $w(T) \leq k$?	a spanning tree T
FLOW	directed graph $G = (V, E)$ and capacity function $c : V \times V \to \mathbb{R}_{\geq 0}$	is there a feasible flow f with value $\|f\| \geq k$?	a flow f
BIPARTITE MATCHING	bipartite graph $G = (A \cup B, E)$ for $E : A \times B$, and weight function $w : e \to \mathbb{R}$	is there a matching S with weight $w(S) \geq k$?	a matching S
STABLE MATCHING	sets M and W of individuals, where $\|W\| = \|M\| = n$, and preferences lists $p_{m \in M}$, $p_{w \in W}$ which are total orderings of the sets W and M respectively	is there a matching of M to W with fewer than k unstable pairs?	a matching S